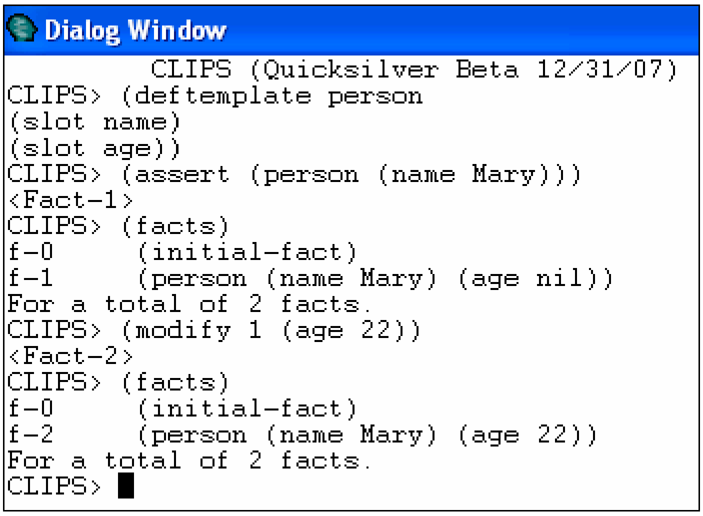
CS 367 Tutorial
11 August 2008
Week 4 (tutorial #2)
Carl Schultz

Material is taken from lecture notes (http://www.cs.auckland.ac.nz/compsci367s2c/lectures/index.html) and the course text book “Joseph C. Giarratano. Expert Systems : Principles and Programming. Brooks/Cole Pub. Co., 1998.”

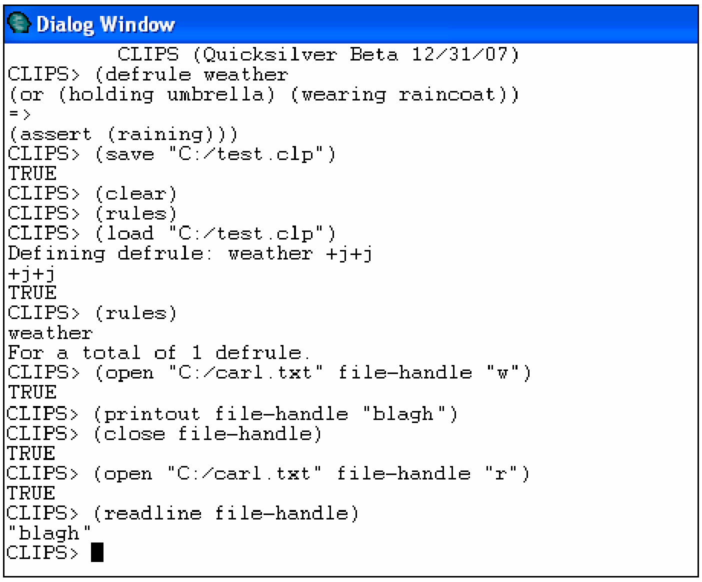
Bits and pieces from last week

- question: can facts be changed?
  - answer: no, facts are static / immutable
  - “modify” CLIPS keyword **retracts** the fact and then **asserts** a new fact with the changes (the id numbers show this – in the example below the fact 1 about “Mary” now becomes fact 2)



```
         CLIPS (Quicksilver Beta 12/31/07)
CLIPS> (deftemplate person
(slot name)
(slot age))
CLIPS> (assert (person (name Mary)))
<Fact-1>
CLIPS> (facts)
f-0      (initial-fact)
f-1      (person (name Mary) (age nil))
For a total of 2 facts.
CLIPS> (modify 1 (age 22))
<Fact-2>
CLIPS> (facts)
f-0      (initial-fact)
f-2      (person (name Mary) (age 22))
For a total of 2 facts.
CLIPS> 
```

- loading, saving and opening
  - load / save are used to manage the CLIPS session (e.g. save the current rules to a file, and then load them back in again later)
  - open is used to manage basic file input / output (e.g. keeping a log of application inputs in a text file)



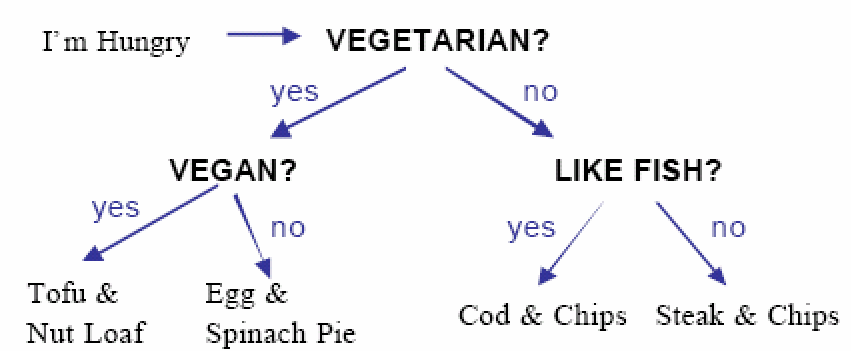
```
         CLIPS (Quicksilver Beta 12/31/07)
CLIPS> (defrule weather
(or (holding umbrella) (wearing raincoat))
=>
(assert (raining)))
CLIPS> (save "C:/test.clp")
TRUE
CLIPS> (clear)
CLIPS> (rules)
CLIPS> (load "C:/test.clp")
Defining defrule: weather +j+j
+j+j
TRUE
CLIPS> (rules)
weather
For a total of 1 defrule.
CLIPS> (open "C:/carl.txt" file-handle "w")
TRUE
CLIPS> (printout file-handle "blagh")
CLIPS> (close file-handle)
TRUE
CLIPS> (open "C:/carl.txt" file-handle "r")
TRUE
CLIPS> (readline file-handle)
"blagh"
CLIPS> 
```

Modelling Knowledge

- we have facts: “You are a vegan”
- we have rules: “Hungry vegans can eat tofu and nut loaf”
- can use rules on current facts to help solve problems, e.g.
    - selection (“I’m hungry – what should I eat?”)
    - diagnosis (“The car won’t start – what is the problem?”)
    - classification (“What animal is this?”)
    - …

Decision Trees

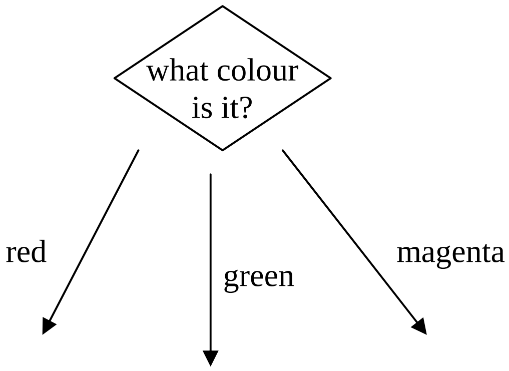
- leaf nodes: solutions to problem, “answer nodes”
- other nodes are called decision nodes



[exercise]
Below are heuristics for choosing which wine to have with a meal:
“If the main course is red meat then serve red wine.”
“If the main course is poultry and it is turkey then serve red wine.”
“If the main course is poultry and it is not turkey then serve white wine.”
“If the main course is fish then serve white wine.”

Construct a **binary** decision tree (yes/no branching) to represent these heuristics.

- decision nodes can have more than two decisions
    - allows for a set of responses, e.g.

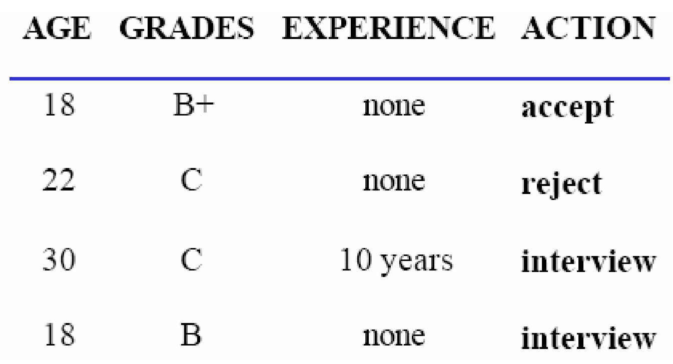


- multiple branching trees can improve efficiency (i.e. questions needed to get to the answer) compared to binary trees. For example, in the above exercise, if "fish" is the main course, then it will take three decisions before deciding on white wine – this can be reduced by using a multiple branch tree.

[exercise]
Using the wine-choosing heuristics above, construct a decision tree with multiple-branches. *Hint:* make the root decision "What is the main course?".

Decision Tables

- we have a set of *n* attributes, e.g. age, grade, experience, …
- each attribute *i* can take one of m_i values, e.g.
    - age: "18", "19", …, "35"
    - grade: "A", "B", …
- each combination of attributes is a *condition* – this is associated with an *action*
- we can make a table of all possible combinations of attribute values



AGE	GRADES	EXPERIENCE	ACTION
18	B+	none	**accept**
22	C	none	**reject**
30	C	10 years	**interview**
18	B	none	**interview**

- use to answer questions: "Should we employ this person?"
- each row is a rule in our knowledge model

Implementing in CLIPS

In general, first thing you need to decide is how to represent knowledge – there are different options available, e.g. could represent served wine heuristics (above exercise) as **defrules** or even just **facts** with **deftemplates** (e.g. animal.clp).

Implementation must reflect underlying data structure. For a decision tree this means:

- implement decision nodes (root and non-leaf nodes)
- implement answer nodes (leaf nodes)

Finally you need to consider:

- handling i/o
- handling application startup / shutdown

Example 1. stove.clp

- mixes i/o and data structure
- decision nodes = CLIPS rules

```
;****************************
;rule20
;     rule to test if a blue tinge on terminals
;****************************
(defrule blue-tinge
 (declare (salience -10))
 (burner problem y)
 (burner type ?) =>
 (printout t crlf crlf
      "Check the ends of the terminals for any sign of a blue" crlf
      "tinge.  Does one exist? y or n ")
 (assert (blue tinge =(read))))
```

- answer nodes = CLIPS rules

```
;****************************
;rule27:
;     handles if there is a crimped wire
;****************************
(defrule crimped-wire
 ?x <- (crimped wire y)
 =>
 (printout t crlf crlf
      "First be sure the stove is unplugged.  Next strip the" crlf
      "insulation away from the crimped area and twist the loose"
      crlf
      "end together.  Solder the wire and cover the splice with" crlf
      "a ceramic nut." crlf
      "*** CAUTION - do not use a plastic nut as the high
      temperature" crlf
      "              will cause it to melt" crlf
      "Afterward, plug the stove in and recheck the element. If
      there" crlf
      "is still a problem rerun this program." crlf)
 (retract ?x)
 (assert (stop)))
```

Example 2. auto.clp

- separates i/o and data structure

  NB: refer to the CLIPS User Guide or the Reference Manual (Volume I) for details about built-in CLIPS symbols and keywords such as $?, read, lexemep.

```
(deffunction ask-question (?question $?allowed-values)
   (printout t ?question)
   (bind ?answer (read))
   (if (lexemep ?answer)
       then (bind ?answer (lowcase ?answer)))
   (while (not (member ?answer ?allowed-values)) do
      (printout t ?question)
      (bind ?answer (read))
      (if (lexemep ?answer)
          then (bind ?answer (lowcase ?answer))))
   ?answer)
```

```
(deffunction yes-or-no-p (?question)
   (bind ?response (ask-question ?question yes no y n))
   (if (or (eq ?response yes) (eq ?response y))
       then TRUE
       else FALSE))
```

- decision nodes = rules

```
(defrule determine-low-output ""
   (working-state engine unsatisfactory)
   (not (symptom engine low-output | not-low-output))
   (not (repair ?))
   =>
   (if (yes-or-no-p "Is the output of the engine low (yes/no)? ")
       then
       (assert (symptom engine low-output))
       else
       (assert (symptom engine not-low-output))))
```

- answer nodes = rule that fires on a “repair” fact

```
(defrule print-repair ""
  (declare (salience 10))
  (repair ?item)
  =>
  (printout t crlf crlf)
  (printout t "Suggested Repair:")
  (printout t crlf crlf)
  (format t " %s%n%n%n" ?item))
```

- …set by previous decision node

```
(defrule determine-sluggishness ""
   (working-state engine unsatisfactory)
   (not (repair ?))
   =>
   (if (yes-or-no-p "Is the engine sluggish (yes/no)? ")
       then (assert (repair "Clean the fuel line."))))
```

- alternative answer node = if no repair was found, system will fall back on:

```
(defrule no-repairs ""
  (declare (salience -10))
  (not (repair ?))
  =>
  (assert (repair "Take your car to a mechanic.")))
```